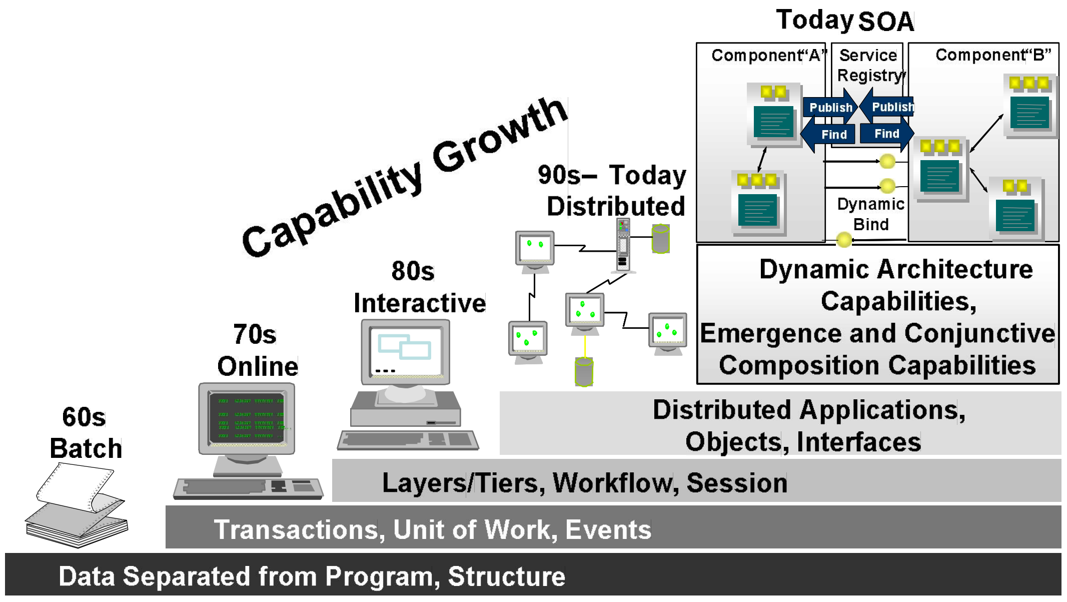
Appendices

APPENDIX A–

Advanced SOA Features

Introduction

SOA has proven challenging because designs must not just satisfy all the capabilities from previous design evolutions (e.g., transactions, sessions, and units of work), but must address a host of new capabilities directly related to SOA: emergence, contracts, cohesion, conjunctiveness, etc.



Today SOA
Component"A"
Service Registry
Publish
Publish
Find
Find
Component"B"
Dynamic Bind
Capability Growth
90s– Today Distributed
Dynamic Architecture Capabilities, Emergence and Conjunctive Composition Capabilities
80s Interactive
70s Online
60s Batch
Distributed Applications, Objects, Interfaces
Layers/Tiers, Workflow, Session
Transactions, Unit of Work, Events
Data Separated from Program, Structure

FIGURE A-1
SOA builds upon all previous architecture evolutions and adds unique features

The Army's operational tempo as exemplified by the Future Combat System requires a dynamic but secure integration of disparate services in both planned and unplanned ways. To accomplish this requires:

- **Dynamic Architecture Features.** These features permit an architecture to be established during execution.
- **Conjunctive Composition Features.** These features enable conjunctive (emergent) compositions, that is SOAs should be designed to provide the ability to use or combine services in ways not conceived by the service's originators.

Dynamic Architecture Features

SOA should deliver services that are a result of dynamic and not static architectures. Traditional architectures are static with component compositions determined at design time. Advanced SOA architectures are dynamic with component compositions necessary to satisfy specific functions or processes determined during execution.

A component is usually a piece of software that has an API and a set of operations on the API that successfully implement a service interface operation. A service, as specified by a service operation, is independent of implementation. A component is best defined in a SOA if it can be autonomously defined and deployed, and relates well to some object in the business.

Dynamic architectures occur when the resolution of what service and component is to be used to contribute to the satisfaction of some business event occurs during system execution as opposed to being pre-designed and statically defined during design or assembly. Traditional systems designs are inherently static in that, minimally, the interface definition used between one component and another is defined during design and is well typed (example: RPC based systems, CORBA Based systems, Internet systems where the IP Address and port of the server is hard coded, or any kind of one-tier or two-tier application).

The value of SOA is the selection of the best possible execution plan, therefore dynamic architectures are required which, in turn, necessitates a specific set of features to be enabled, which are included in the following table:

TABLE A-1. Features Necessary to Enable Dynamic Architectures

Feature	Description	Why Related to Dynamic Architecture	Examples of How Enabled
Late binding	Service collaborations resolve which components will be used to satisfy a service request as late as possible during execution. It is the result of • Selecting a service by specifying a service interface that satisfies the consumer need as late as possible during execution and then, • Resolving the interface to a specific component and the component API that best satisfies the service interface signature also as late as possible. Each binding should be considered a configuration that can change dynamically as needed without loss of correctness and be consistently applied regardless of the enabling environment.	Because the choice of which service, which operation, and which enabling component should occur during execution.	Usage of Service Proxies to resolve the connection of one service to another. Initially these proxies may use configuration files to pass the fully resolved location of a service. However as an Enterprise Service Bus (ESB) is implemented, the proxies may be changed to call one or more registries managed by the ESB.

Feature	Description	Why Related to Dynamic Architecture	Examples of How Enabled
Loose coupling	The low degree of mutual interdependence between components enabled in a SOA. It is the result of • Rigorous encapsulation of data around components implementing services • No foreign keys existing between service domains • Application specific metadata being applied to the service during execution • Stateless execution contexts • Self-description of message contents • Moving away from synchronous, RPC-like messages to asynchronous messages. It also means that the enabling components are autonomously deployable and have minimal to no shared data.	Because hard-coded relationships limit late binding and autonomy.	• Usage of Data adapters to access the data (encapsulation) • Usage of SOAP-based messages to encapsulate protocol implementations (Semantic data independence) • Generalized Metadata repositories used to integrate information together semantically
Discrete functions	The specification of each method of a component is independent of all other methods. It results from designing, or confirming that COTS products contain, component methods, APIs, and the corresponding service interfaces that are autonomous and independent, do not presume any application context, and are stateless between invocations. The primary success measure is to be able to use each method in many scenarios with minimal preconditions. Note that a method is a specification that ultimately results in the definition of an operation on an interface. You can think of a method as the specification for a service interface, whereas the operation is the specification of the operation enabled by an API that uses a specific port number that corresponds to a type of interface. This allows for late binding of the specific operation being used to satisfy a specific service method.	So you can add context-specific information at execution time. You use a combination of late-binding meta data and implementation selection rules to limit the number of deployed implementations and improve context sensitivity.	• Implementation planning at the Interface (operation) level. This means that the primary output of a delivery (e.g. of a Sprint 30 day delivery cycle) is operations to be enabled. All the other “plumbing” is encapsulated and facaded under the interface • Each operation is autonomous, therefore can (1) registered by a Service Registry, (2) have its own evolution plan

Feature	Description	Why Related to Dynamic Architecture	Examples of How Enabled
Service Discovery	This is a process of querying, browsing, offering, and selecting specific instances of operations on interfaces (services) to satisfy a specific request. It is the result of using an interface registry to provide the ability to automatically identify hardware or software services and then bind to them to perform some necessary piece of work. In its simplest form of service discovery is the result of using a configuration file that removes any hard coded relationships between components. In its most advanced form it means registering ontology of characteristics that can be queried to determine whether a specific operation on a specific interface will satisfy a specific need. The primary success measure is • To never statically define the association between components in code, but rather resolve what specific component(s) are to be invoked • To provide a service by dynamically determining what interface is to be used and then invoking an enabling component through a selected API.	Because if you do all the above, you now need a way of discovering the best possible service, operation, and discrete function (component) to use.	In the short term, Service Proxies to configuration files/databases can be used. Long term, proxies can enable "behavioral browsing" of the service registry to assure that the best possible fit for an operational need is satisfied with the most correct interface(operation)
Autonomous deployment	This is the ability to deploy components and interfaces into production with minimal constraints on how, when, or how often the deployment occurs. It is the result of using loose coupling, late binding, discrete functions, and service discovery to enable components to have minimal to no prerequisite external or fixed dependencies. If services are statically bound to specific components, then fixed dependencies exist. If components are packaged into static products, then other fixed dependencies exist. And if components have mandatory and prerequisite associations with other tools or components, then yet more kinds of fixed dependencies exist. Fixed dependencies reduce, or even eliminate autonomous deployment and the ability to form dynamic applications.	Autonomous deployment allows services to be added continuously by different service providers. Services can be selected as late a practical during execution instead of the previously deployed services because the selection of a service in a dynamic architecture is based upon a set of conditions instead of specific implemented service.	Usage of Service Delegates, Facades, and Proxies to preserves the independence of the Interface (operation) while at the same time enabling Services with Open-Source and COTS solutions.

Feature	Description	Why Related to Dynamic Architecture	Examples of How Enabled
Message-Based Integration	This is the usage of event-generated messages from autonomous components to dynamically form application contexts. As opposed to statically defined applications that have fixed application contexts and fore-knowledge of what each component of the application does, message-based integration relies on the messages to provide application context. For messages to provide application context event brokers must be used to generate and type events, taxonomies and semantics must be portable and not based upon a single shared database, channels for routing messages from consumers to providers must extend across the COI, dispatching and translation must occur from one environment and domain combination to others, and replicated service directories within each environment/domain combination must exist to qualify which specific set of services should contribute to a specific application context. In an extended or inter-enterprise environment federated service design plus an especial emphasis on dispatching and translation services is necessary to enable the formation of dynamic applications across enterprise boundaries.	Because this is how service vectors are now constructed. Besides, it's necessary for autonomous deployment.	• Standard SOAP based message formats for the data packets being passed from component to component. • Application of ontologies coupled to role-based access to assure that the right information is passed to the right service to satisfy the information request needs. This is important because a piece or data (or a service interface) can have different meanings in different ontologies. • Usage of 'deep crawling' of complex document/record/transactional data to create information indexes. Note that deep crawling is a search concept key to SOAs. It comes from deep crawling of the internet in which a bot or crawler examines not just the header information of a document, but its contents and, if allowed, the data in databases. The result is a taxonomy tree, and in some cases, an ontology.
Independent and implementation-neutral	This means not associating a service with a specific set of programming languages or implementation platforms. However, the characteristics of various enabling environments and programming languages require that the service, as specified by a service contract specifying an instance of a service interface, be specific to all environments of a Communities of Interest (COI). This apparent contradiction means that • Specific environmental implementations of interfaces and operations must exist in all environments of the COI and, • Minimally, requests are routed to the enabling environment of the component that implements the service, or maximally, the enabling component is uniformly implemented in all environments of the COI.	Because you don't want to limit from whom or where, or with what the discrete functions are provided.	• Ensure that interfaces can be used in relevant environments and securely dispatched to all information enclaves. • Proactive and advance usage of Security Assertions Markup Language (SAML). SAML is a mechanism for authentication and authority, its usage with service based messages is the mechanism by which trust of the message is asserted. If you can't do this, then identity implementations are really-really tough.

Feature	Description	Why Related to Dynamic Architecture	Examples of How Enabled
Coarse grained components	This is the design and selection of enabling components to represent classes of business entities as opposed to specific instances, which results in better alignment of the mission to the software services. It also reduces dependencies among participants and reduces communications to fewer messages of greater significance.	Because to do otherwise floors the system and destroys the registry.	• Target Federated Enterprise Architecture and Target Solution Architecture Lifecycle stages provide models of business components to be enabled. • Where applicable, the product-centric nature of the business component definitions in our implementations should be preserved. A crude way of saying that business components should be autonomously deployable in the same way that any product is autonomously deployable. • A distinction should be made between sharable business components that may be applicable to two or more business-centric components (e.g., Identity Management is applicable to Information Discovery and Portal Services, equally). Two problems here: Think of the Spell Checker embedded in Microsoft office: (1) is it a coarse-grained component that can be installed independently of office, and (2) how do we assure that is applicable to Excel, Word, Access, and PowerPoint? Is it trustworthy to use the Office Spell-check in, say, IBM Notes? The point here is that a coarse-grained component may or may not be designed to be incorporated into other components during design or assembly time as opposed to runtime through a Service Registry (the stereotypical approach to enabling service-ness). Who authenticates and assures this assembly
Dynamic service collaborations	This is the basis for forming processes and composite services. Dynamic service collaborations will use either Orchestration (e.g., BPEL) to enable processes to satisfy scenarios or Choreography (e.g., WS-CDL) to coordinate the execution of composite services.	Because higher-level services and all business processes are all collaborations constructed for the purpose of responding to specific event types	Usage of Orchestration (BPEL) to enable processes and Choreography (WS-CDL) for composite services.
Synchronous, asynchronous, and publish-subscribe interactions	All should be supported with an emphasis on asynchronous and publish-subscribe interactions, which are especially important for the inter-enterprise and multiple information domains that represent the Army environment.	Because a dynamic architecture may require different levels of service.	Mediated, non-mediated synchronous and asynchronous messaging with uniform and common message bus adapters and proxies.

Conjunctive Composition Features

SOA should deliver services that are a result of conjunctive composition. That is, SOAs should be designed to provide the ability to use or combine services in ways not conceived by the service's originators. Conjunctiveness is achieved by applying late-binding and application specific metadata to the service. Key aspects of conjunctiveness are (1) services interact with each other with minimal preconditions, and (2) they can evolve to meet specific needs without jeopardizing existing application contexts. Table A-2 includes specific features that enable conjunctiveness.

TABLE A-2. SOA Features Enabling Conjunctiveness

Feature	Description	Examples of How Enabled
Service contracts	Detailed specification of actions to be performed and data to be provided by service providers for specific types of service consumers. Service contracts should have very detailed signatures, resolve the association between a community of interest and a particular service interface given the application context of the late binding service request. SOA changes the power equation of systems delivery because consumers of services can reject binding to a particular instance of a service in favor of a competing service instance if the 'fit' is not correct. Therefore, service contracts represent the basis for finding and then delivering a service that 'fits'.	Metadata defined application contexts represented by model for information collection and COI specification.
Dynamic Process Formation	Processes are created as dynamic service collaborations (above) that result in time-dependent sequences that satisfy specific transformational objectives.	Dynamic (temporal and event driven) as well as pre-defined process support via metadata description,
Composite services	Functions are enabled as composite services that are associations of services in a stateless and peer-to-peer collaboration to satisfy a capability-based set of services (e.g., a collaboration service is really a dynamic collaboration of a virtual space, chat, log, presentation, and other lower-level services).	Best-practice formation of services to deliver IDP-specified capabilities
Application-neutral design	Means that service should not be designed to presume a specific application context. A real example of this comes from a manufacturing company that had a common shopping cart site. The problem was (is) that each product line had its own pricing calculator. The application context was embedded. This made it very difficult to implement company wide policies. This company should have designed a single, application-neutral calculator and, using metadata, bound the application context based upon the instance of usage.	Usage of Facades, Delegates, Factories, and Containers; Interface Facades brokers.
Consumer and Provider Metadata	Consumer must be expressed using a consistent metadata model, and provider offerings should be expressed using an equally consistent metadata model.	Session objects of metadata. Ontologies registered in service registry, Metadata usable by all services.
Concurrently developed	Service interfaces and components are developed and deployed without preconditions. This reinforces the need for independent and discoverable services and application neutral design techniques.	Value-drive and feature based SPRINT planning with concurrent execution.

Feature	Description	Examples of How Enabled
Interoperable across a heterogeneous environment	Results from managing identity, method exchange, event preservation, and syntactical and semantical data exchanges. The objective is for two or more components to not only exchange and use data but to discover and then accept/post operational requests to components irrespective of how or where the components are enabled. This requires across the domains of a COI: • Identity management of components and executing instances. • Platform-neutral formats for data, method, and events. • Exchanges and method posting/acceptance must exist across the network addressable space. and • Community of interest specifications controlling access to interoperable services.	Usage of Service Proxies, Adapters, Dispatching Services between enclaves, SAML-based entitlement and attribute access; dynamic and static COIs.
Reusable artifacts	Whereas autonomy is a measure of dependencies, reuse is a measure of shared elements. The minimal requirement for SOA is interface reuse where a service interface's signature can enable numerous components. The interface is reused with the service contract specifying the basis for choosing one component over another.	Support for design time reuse (e.g., shared library of source and linkable objects as well as runtime reuse by the delivery of autonomous services.
Network addressability	Means anyone, or any component, that is anywhere in the network addressable space should be able to participate in a service collaboration – given appropriate security constraints. By using Web Services-based standards the network addressable space is extended to the entire Web	Federated and replicated registries across domains; Dispatch services; Proxies for cross domain services access.

APPENDIX B –

Advanced SOA Delivery Tactics

Introduction

This appendix describes various techniques and delivery tactics. The objective of this appendix is to describe advanced aspects of SOA based delivery and its impact on the methodology.

SOA Software Engineering

SOA Software Engineering is, out of necessity, different from traditional software engineering because the Dynamic and Conjunctive features (Appendix A) of a well formed SOA requires that different approaches be used to assure that the Technical Performance Measures (TPMs) for each feature have been achieved and enabled. Consequently, the software engineering process has to be especially tuned for successful SOA delivery. Unlike traditional top-down and incremental software engineering, engineering for a SOA is iterative, continuous, requires additional views of the architecture, and places an extra burden on the governance process. These differences have been summarized in the table, below.

TABLE B-1. Comparison of Traditional and SOA Software Engineering

Method / *Factor*	Traditional Software Engineering	SOA Software Engineering	Comments
Delivery	Self-contained program-based delivery (e.g., program silos). Self-contained programs with point-to-point interfaces to other programs; minimal cross-program sharing.	Marketplace based delivery. Small deliveries organized around groups of services enabled by autonomously delivered components with the expectation of reuse in multiple problem contexts.	The primary driver of traditional software engineering is deliver an autonomous system with little or no expectation where all the components and data are tightly bound to the program delivery specifications. The driver for SOA software engineering is deliver reusable services into various problem domains.

Method / Factor	Traditional Software Engineering	SOA Software Engineering	Comments
Delivery Organizations	Traditional Program Team structure (e.g., pre-allocate all resources to specific task orders at beginning at project) Traditional Roles (e.g., developer, coder, architect, project manager, etc.)	Associative and adaptive structures (e.g., pulling resources from a pool and continuously allocating and reallocating them to the needs of a specific delivery.) Uses similar roles, but developers and coders must now accommodate the need for the assembly of pre-existing services and delivery of new services. Also, data architects / developers have to accommodate the need for payload design (i.e., the design of the interface operation's signature) and the segmentation of the data to accommodate service delivery.	New Skills-- • Creation of individual services • Orchestration of services into business processes • Assembly of services into composite applications • For the same components and/or services the management of overlapping requirements, development, delivery, and support.
Methodologies	Structured Analysis & Design Object Oriented Analysis and Design (OOAD) can be used if desired	• OOAD or Model Driven Architecture (MDA) methodology is desired and improves the results • Feature Driven Development (or Similar)	• It is actually very hard to do a SOA with Structured Analysis and Design because design is not just top-down, and data and process need to be designed together.
Software Development Lifecycle	Waterfall, Incremental, or Prototype based	• Agile Alliance • SCRUM, or • Crystal based	• Service Components can continuously evolve. Iterative (not Incremental), continuous, evolutionary
Software Architecture Engineering	Structured Decomposition Data Flow Diagram (DFD) based	• Need additional views and more formal associations between the views that show how different aspects interact with one another.	• SOAs demand additional views; interface, component, and deployment views are necessary.

Method / Factor	Traditional Software Engineering	SOA Software Engineering	Comments
Integration	Point-to-Point	• Dynamic and Static Interfaces • Discovery Mechanisms	
Verification & Validation	• Assumes a well-established requirement base that is traceable and transformed from the start of a project/program all the way to the end. • Assumes deterministic computing (e.g., predefined relationships between artifacts established at design time).	• Iterative addition of detail • Requirements don't have to be 'done' to start fielding solutions. • Verification is the traceability of one or more requirements to an enabled feature. Much more granular than traditional software engineering. • Non-deterministic (e.g., relationship between artifacts is 1) not completely known until a particular event occurs during execution and 2) can change over time). • Orthogonal Array Testing Strategy (OATS) helps identify the important service-service interactions and derive a minimal set of test cases. • Need to test initial delivery of services and test again when they are applied in different problem contexts.	• SOA components are never "finished, therefore verification is also iterative. • Especial attention must be paid to testing for "unknown" customers; that is, those users of a service 'sometime later' that are not known at the time of initial service delivery. • Reuse of a service requires the publication and usage of certification suites to confirm that the reused service behaves consistently in all contexts
Cost Recovery	• Simple: Pay by-the-Keg	• Difficult: Pay by-the-Drink • Managed services and services on demand	• SOA fosters reuse, and reuse means complex cost recovery schemes.
Governance	• Business Area/System Based • Typically, governance enabled for projects or programs, not the enterprise. • Governance 'tuned' to the SILO-based style of delivery.	• Enterprise Based • Associative and not hierarchical • High degrees of overlap of system development life cycle phases • Manage autonomy and concurrence of delivery.	• SOA needs more formal and complex Governance

All of the above differences in software systems engineering can be traced to two sources:

- Changes in the model that defines a SOA; that is, the characteristics of a SOA that make it distinct from other architecture styles. To address this, the Object Management Group (OMG) is establishing a UML Profile for SOA.
- The need to control, execute, and manage the delivery process so that the benefits of loose coupling, autonomous deployments, and the other features reference in Appendix A are realized. To address this, a different way of delivery is required that is based upon concurrent and continuous delivery, for example, use of best practice such as Information Technology Infrastructure Library (ITIL).

Each of these, the UML Profile for SOA, and Continuous and Concurrent delivery will now be discussed and foundations for advanced SOA adoption.

Establishing a UML Profile for SOA

Currently, there is no accepted standard regarding what attributes should be included in a SOA design. Therefore the question of whether a particular design and implementation is “good” or “bad” is often left to the individual criteria of those charged with delivering or judging a SOA. Therefore, the lack of a consistent model of SOA will necessarily mean that services designed by one ‘faction’ of delivery will embody different characteristics and features than another “faction.” This has made conjunctive integration difficult.

The Object Management Group (OMG) is currently working on a ‘UML Profile’ for SOA that will provide an objective basis for specifying the various attributes in a SOA design. The resultant SOA Specification will exhibit high conceptual integrity for the initial specification, delivered software and services, and managed SOA compliant solutions. Characteristics of this UML framework include Views and specific UML Profiles Models for each View. Together, they represent what needs to be done to deliver functional requirements in a SOA way. They are also used as the basis for specifying SOA design to implementation artifacts.

Views

Architecture views are nothing new. The Department of Defense Architecture Framework (DoDAF) is based upon a set of views. So are the Federal Enterprise Architecture (FEA) and the Enterprise and Rational Unified Processes. What's different is the number and the associations between views so that dynamic and service-based deliveries can be well described.

The figure, below, describes a necessary set of architecture views as a set of associations as opposed to, as in traditional engineering approaches, as a stack of some sort (example: the FEA Service Model). To be successful in delivering SOA means that many activities have to go on at the same time. For example, requirements for both the consumer and the provider mature, and the providers are delivering new versions of services, and autonomous deployments occur across the enterprise, etc. To enable this concurrency across the Army enterprise, the views must be defined as having explicit association with each other. This will then assure that effective input/output relationships, traceability of an artifact from one view to another view, transformation from concept to deliverable, and control of the total process can be achieved.

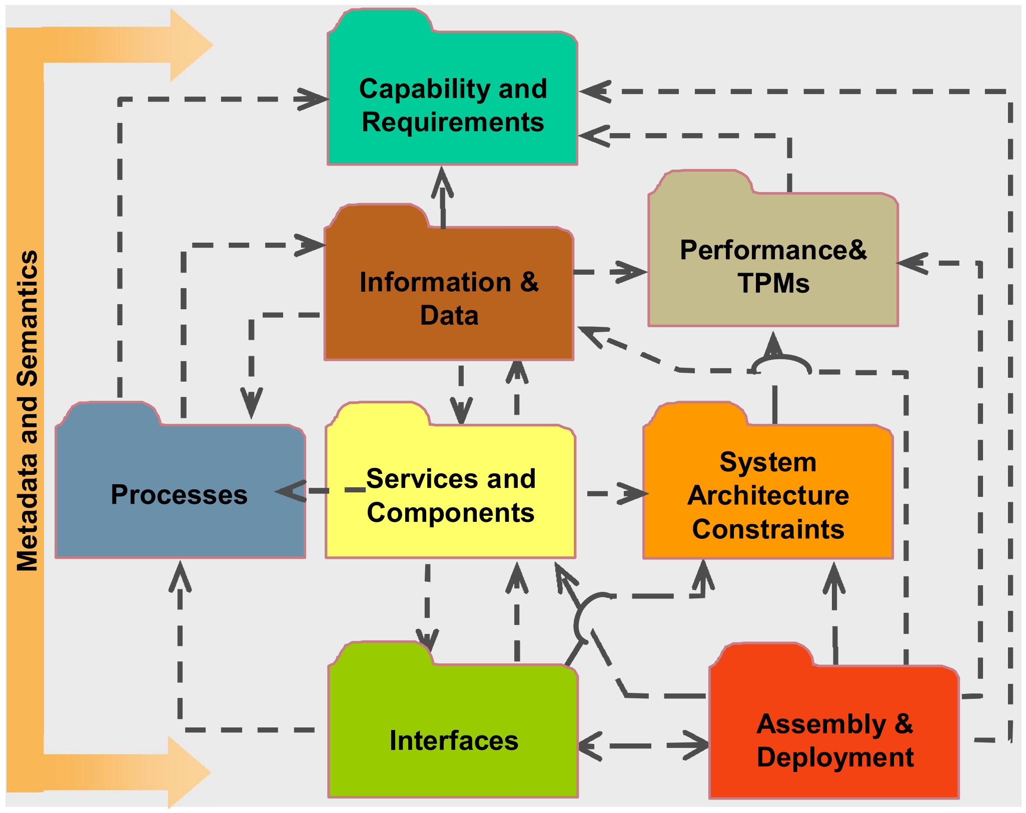


FIGURE B-1
Associative Model of SOA Architecture View Set

A short description of each of the above architecture views follows—

1. **Capabilities & Requirements:** Establishes measurable and modeled descriptions of all capabilities, desirable functions, SLA's, and constraining requirements.
2. **Business Processes:** Establishes detailed 'stories' and use cases. This is the basis for featurization.
3. **Semantics & Metadata:** Describes the communal and contextual 'facts' that the various products to be delivered are based upon.
4. **Performance & TPMS:** Establishes measurable and modeled 'ilities' that drive implementation architecture and deployment decisions.
5. **System Architecture Constraints:** Translates the Performance and TPMS into implementation architecture decisions: this is the place that pattern-based decisions on how a specific feature or set of features will be implemented.
6. **Information and Data:** Uses information from Capabilities, Business Processes, and Performance and TPM views to establish data payloads for interfaces plus how data will be consumed and used. Includes service-oriented segmentation rules.
7. **Interfaces:** Establishes interfaces and operations for specified products that invoke various implementations. Promotes service-oriented 'abstraction' by

keeping interfaces and operations constant and allowing the implementation(s) to be chosen to enable specific needs.

8. **Services and Components:** Establishes the component and inter-process descriptions that will drive deployment descriptions. For example, should component 'A' and 'B' be compiled together during design time, linked together at build time, or use an interface registry to communicate with each other?
9. **Assembly and Deployment:** Establishes the model of how the products and resultant implementations should be constructed for deployment.

Elimination of Ambiguity: The Use of UML Profiles for Each View

Specifying that there are a number of architecture views is necessary but not sufficient to promote a Service Marketplace. Enablement and delivery of services have to be based upon a consistent and measurable standard. The standard should assure the integrity of the service to production in such a way that it can be used and reused. This level of measurable certainty is the basis for

- Making the delivery of SOA less dependent upon individuals and more upon the specification of what represents good and bad design,
- Software, Information, and Mission Assurance.

Consequently, the UML Profile for specifying what a SOA is, and by extension, how the absence or presence of any one characteristic of SOA should be addressed, is the basis for a consistent specification of the conceptual integrity of SOA.

Below is an example of part of a UML Profile for one of the SOA views.

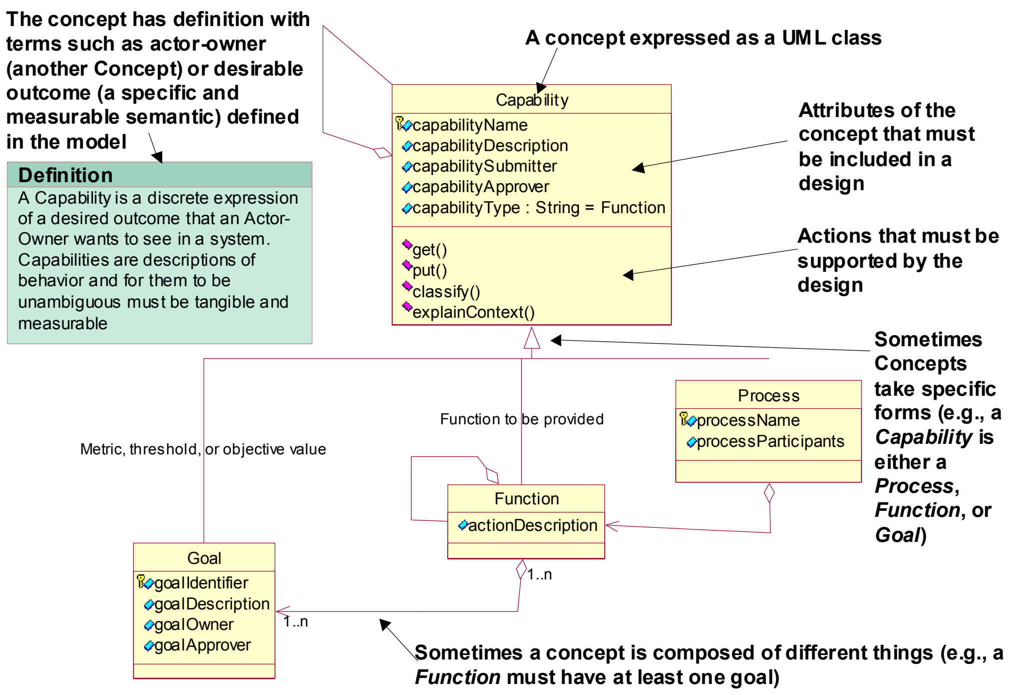

The concept has definition with terms such as actor-owner (another Concept) or desirable outcome (a specific and measurable semantic) defined in the model
Definition
A Capability is a discrete expression of a desired outcome that an Actor-Owner wants to see in a system. Capabilities are descriptions of behavior and for them to be unambiguous must be tangible and measurable
A concept expressed as a UML class
Capability
capabilityName
capabilityDescription
capabilitySubmitter
capabilityApprover
capabilityType : String = Function
get()
put()
classify()
explainContext()
Attributes of the concept that must be included in a design
Actions that must be supported by the design
Sometimes Concepts take specific forms (e.g., a Capability is either a Process, Function, or Goal)
Process
processName
processParticipants
Function to be provided
Metric, threshold, or objective value
Function
actionDescription
1..n
Goal
goalIdentifier
goalDescription
goalOwner
goalApprover
1..n
Sometimes a concept is composed of different things (e.g., a Function must have at least one goal)

FIGURE B-2
SOA UML Profile Snippet

Adoption of a UML Profile for SOA will ease the management, control, and inherently chaotic delivery to production of services by providing a basis to not only assure that continuous and concurrent delivery can occur, but that what is developed and delivered can be effectively measured as being viable.

Continuous and Concurrent Delivery

If a UML Profile for SOA provides the conceptual integrity for how to develop and deliver SOA solutions in a predictable and measurable way, then Continuous and Concurrent Delivery uses the views and associated UML profiles to deliver on the promise of a SOA Marketplace in which services are delivered to production consistent with the operational tempo and the every-changing operating environment of today's Army. For example, a feature-based planning activity feeding a SPRINT[1]-like delivery process can generate deliverables every 2-4 weeks across multiple delivery teams. And, because feature-based planning is reflected in the Integrated Master Schedule, complete transparency is achieved with all Stakeholders.

Feature Based Planning for a SOA

A feature is a single, well-formed enabled outcome. In software, it is usually associated with a single event invoking a specific set of rules resulting in a single post-condition state. Therefore, for a SOA, a feature is associated with the specification of an interface and a specific operation on that interface and can be

[1] 'SPRINT' is a part of an open and agile development process called SCRUM (http://www.scrumalliance.org/). SPRINTS call for delivery of features every 2 weeks to no later than 30 days. The objective is to deliver 'inside the requirements change cycle.'

considered the implementation of a service. Implementing a service is different from implementing the enabling component that is used for a service.

- Features represent specific tasks of specific use cases to be performed.
- By understanding the importance of one use case over another, we can understand the value proposition of a specific feature. This allows a more "surgical" delivery process.
- It is easier to replan a feature than it is a tightly coupled system or subsystem.
- There are fewer dependencies between other features and on resources if planning is performed at the feature level.
- Planning can be based on the value of individual features as opposed to tightly bound phase deliveries.

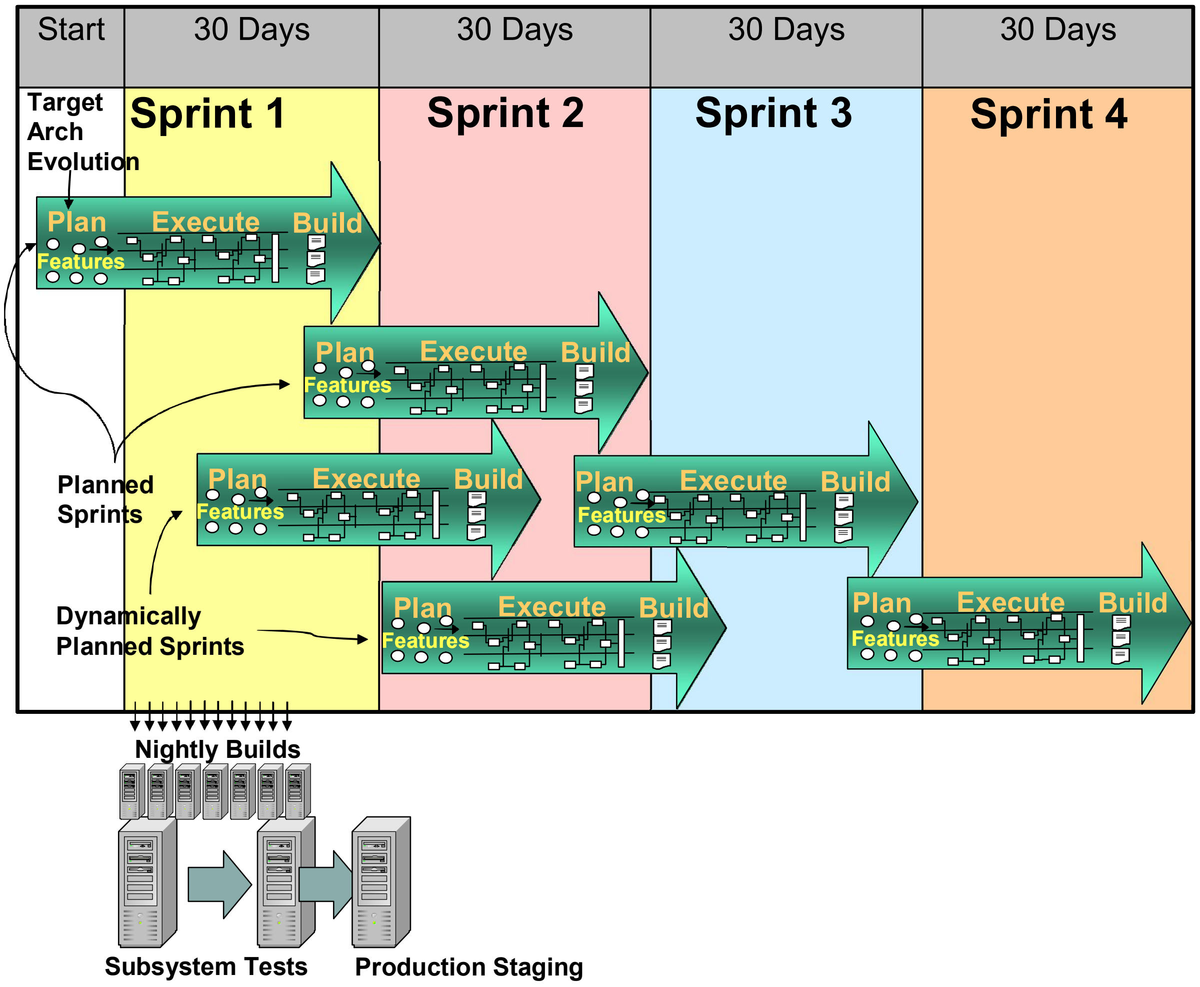


FIGURE B-3
Plan and Deliver by Feature

Controlling and Managing SOA Delivery by Feature

The above figure depicts the delivery by multiple autonomous delivery teams. The problem is that this kind of delivery is easy to depict and hard to do without an appropriate framework managing the process. The figure below depicts an example of a ITIL-based process by which appropriate planning and control of all delivery, new or maintenance driven, is managed by the same process.

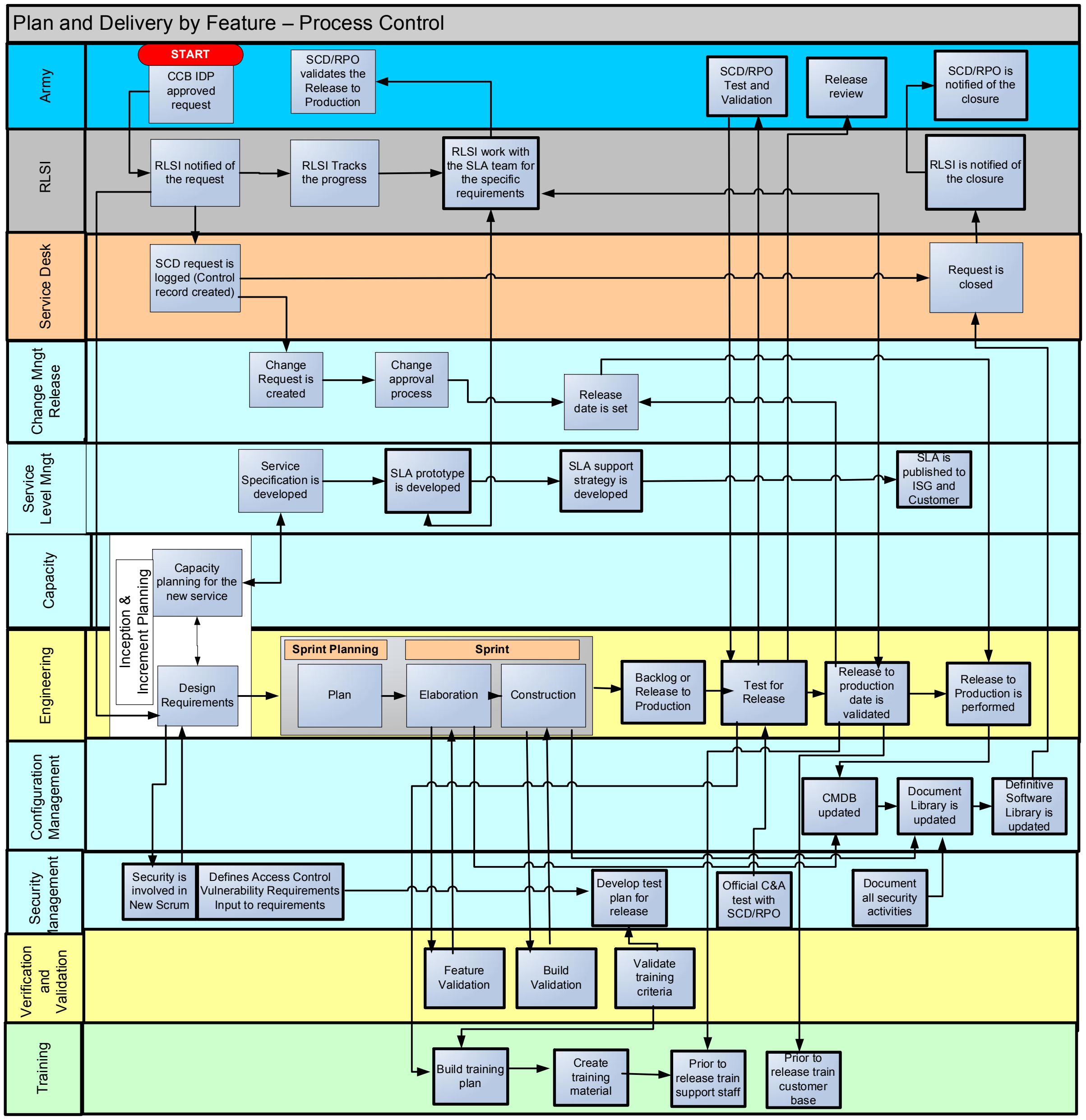

Plan and Delivery by Feature – Process Control
Army
RLSI
Service Desk
Change Mngt Release
Service Level Mngt
Capacity
Engineering
Configuration Management
Security Management
Verification and Validation
Training
START
CCB IDP approved request
SCD/RPO validates the Release to Production
SCD/RPO Test and Validation
Release review
SCD/RPO is notified of the closure
RLSI notified of the request
RLSI Tracks the progress
RLSI work with the SLA team for the specific requirements
RLSI is notified of the closure
SCD request is logged (Control record created)
Request is closed
Change Request is created
Change approval process
Release date is set
Service Specification is developed
SLA prototype is developed
SLA support strategy is developed
SLA is published to ISG and Customer
Inception & Increment Planning
Capacity planning for the new service
Design Requirements
Sprint Planning
Sprint
Plan
Elaboration
Construction
Backlog or Release to Production
Test for Release
Release to production date is validated
Release to Production is performed
CMDB updated
Document Library is updated
Definitive Software Library is updated
Security is involved in New Scrum
Defines Access Control Vulnerability Requirements Input to requirements
Develop test plan for release
Official C&A test with SCD/RPO
Document all security activities
Feature Validation
Build Validation
Validate training criteria
Build training plan
Create training material
Prior to release train support staff
Prior to release train customer base

FIGURE B-4
Planning and control of a SOA-based Concurrent and continuous delivery (example)

Delivering Individual Sprints

Finally, within the overall ITIL-framework for continuous and concurrent SOA-based delivery are the specific details of the engineering delivery team. Depicted in the figure below is a typical feature driven delivery process.

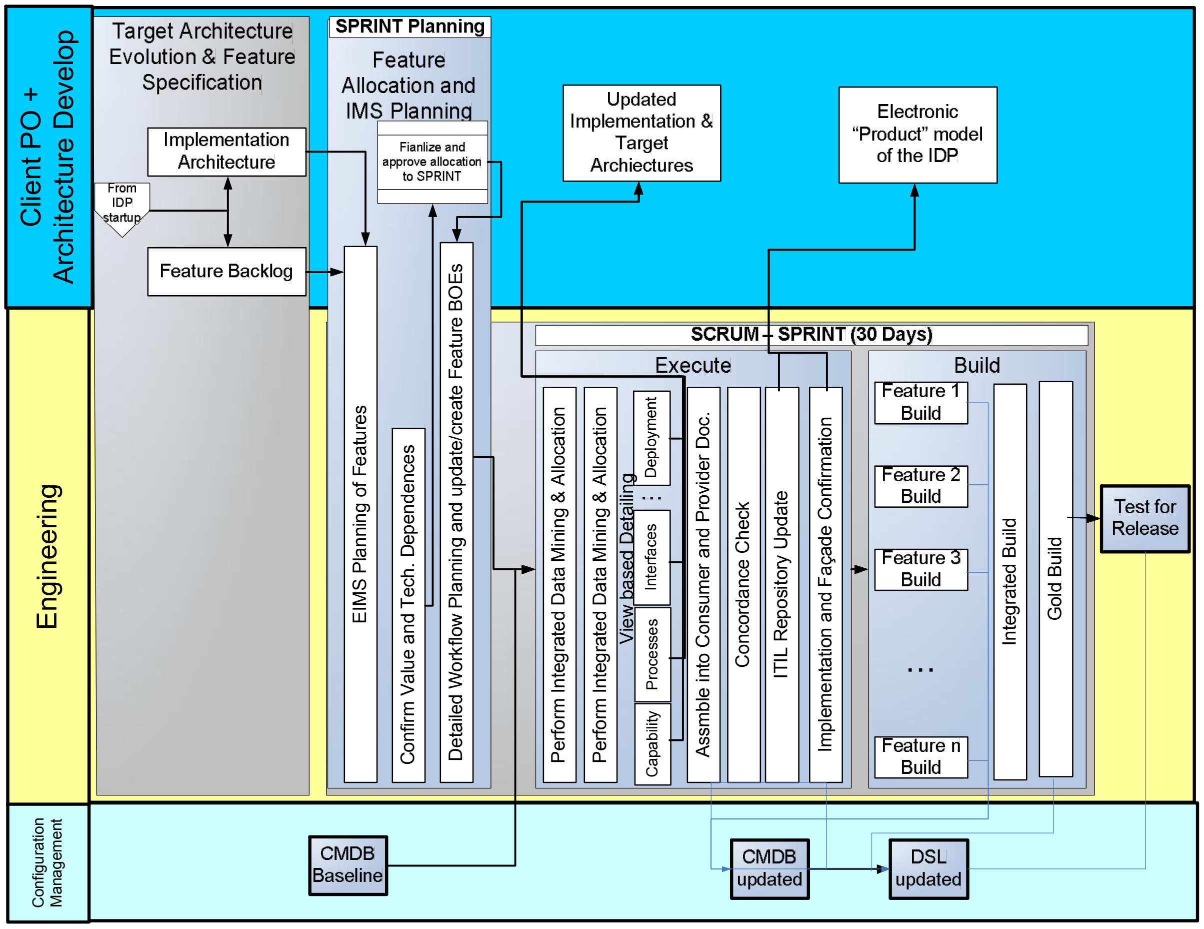

Client PO + Architecture Develop
Engineering
Configuration Management
Target Architecture Evolution & Feature Specification
Implementation Architecture
From IDP startup
Feature Backlog
SPRINT Planning
Feature Allocation and IMS Planning
Fianlize and approve allocation to SPRINT
EIMS Planning of Features
Confirm Value and Tech. Dependences
Detailed Workflow Planning and update/create Feature BOEs
Updated Implementation & Target Archiectures
Electronic "Product" model of the IDP
SCRUM – SPRINT (30 Days)
Execute
Perform Integrated Data Mining & Allocation
Perform Integrated Data Mining & Allocation
View based Detailing
Deployment
Interfaces
Processes
Capability
Assmble into Consumer and Provider Doc.
Concordance Check
ITIL Repository Update
Implementation and Façade Confirmation
Build
Feature 1 Build
Feature 2 Build
Feature 3 Build
Feature n Build
Integrated Build
Gold Build
Test for Release
CMDB Baseline
CMDB updated
DSL updated

FIGURE B-5
The Engineering of a Specific Sprint

APPENDIX C –

SOA Security and Information Assurance

Introduction

Assurance, in general, measures the degree of confidence in the trustworthiness of an entity in some system context. *Trust* is established based upon sufficient and credible *evidence* leading to belief that the entity in some system context satisfies the specified requirements. With that a basis,

> *Information Assurance* refers to the ability to access and defend information and preserve the quality and security of that information. This involves measuring and assuring information availability, integrity, authentication, confidentiality, and non-repudiation. It also includes providing for restoration of information systems by incorporating protection, detection, and reaction capabilities.

The above definition was based upon the published definition of Information Assurance by the Information Assurance Directorate of the NSA.

The Problems SOA Introduces

It would be tempting to simply adopt the above definition for SOA, but the problem is that SOA adds a couple of things that strike at the foundation assurance, therefore at the foundation of information assurance.

What are the requirements?

A foundational tenet of assurance is that there are requirements that can be tested and that confidence and trust in the satisfaction of those requirements can be built in and measured as having been satisfied. What, exactly, are the requirements for a SOA?

- The ones that existed when a service was built and delivered?, or
- The ones used to decide that it should be reused in an unplanned for process or function months or years after delivery?

Services are initially defined and deployed based on as set of requirements. In traditional systems those requirements control the usage of the software. But for SOA where services can be selected for usage days, weeks, months years after initial deployment there is no control over whether or not the user's requirements are consistent with the requirements that drove the service enablement.

What is the system?

Another foundational tenet is that requirements are satisfied in the context of a specific system. What, exactly, is a System in a SOA environment? Since a foundational design tenet of SOA is that systems are dynamic and determined during execution, then what is the credible evidence that a constantly forming and reforming 'vector' of services in response to specific events is assuring *"... availability, integrity, authentication, confidentiality, and non-repudiation and that it is providing for restoration of information systems by incorporating protection, detection, and reaction capabilities*"?

With this as a backdrop, this appendix explores SOA Security and IA. Specifically, it explores:

- **The Impact of SOA on Security and Assurance.** This subsection establishes a discrete basis for the differences to be accommodated in a SOA IA strategy
- **Extending the SOA UML Profile for Security.** This subsection discusses extending the SOA UML Profile overviewed in Appendix B to include security and assurance parametrics and models.
- **SOA Security and Assurance Methodology Impacts.** This subsection discusses the impact of SOA Security and Assurance on a SOA methodology. If the essence of SOA is dynamic and conjunctive composition, what are the impacts on a SOA methodology that has to assure that Security and Assurance are built in?

The Impact of SOA on Security and Assurance

Appendix A described specific features necessary for enabling dynamic but secure integration of disparate services in both planned and unplanned ways. This section examines the Security and IA Challenge that results from enabling each SOA feature.

Impact of Dynamic Architecture Features on Security and IA

The enabling of features necessary for Dynamic Architectures challenges the nature of a systems context. Since a "system" is a dynamic composition of services derived during execution regarding both the class of a service to be chosen as well as the specific instance of that class to be invoked, this results in the foundational question—

> *What metric or monitoring can provide an indication of how well I'm securing the SOA?*

Further, each feature described in appendix B results in its own set of IA challenge to providing a secure and information assured environment

TABLE C-1. Dynamic Architecture Security and IA Challenges

Feature	Security and IA Challenges
Late binding	Processes are formed dynamically therefore never completely sure what services will participate
Loose coupling	Loose coupling desirable for good software design, but tight coupling may be necessary for maximum performance. Finding the right balance is important.
Discrete functions	Designing for reuse of independent methods requires additional attention and care. What references can be used to certify and accredit SOA services before they are added to ensure adequate security has been addressed?
Service Discovery	Varied techniques offered and planned for, such as: directory services, service brokers, service auctions, service policies, and Quality of Service (QoS) criteria, including reliability, performance, availability, and trustworthiness.
Autonomous deployment	Component "hot swap" requires sophisticated software and hardware design that does plug-and-play.
Message-Based Integration	Focused on correlating simple and complex relationships of events based on past trends and future predictions. Must react to new, external input arriving at unpredictable times. How will data encryption be handled in the service-oriented architecture?
Independent and implementation-neutral	Avoiding implementation pitfalls and good practices for a clean separation of concerns
Coarse grained components	Requires proper domain analysis of business processing and ontology.
Dynamic service collaborations	Orchestrate and "team" software processes to solve problems collaboratively or compete intelligently. How will SOA security 'fail over' for those services that don't/can't react to the SOA paradigm?
Synchronous, asynchronous, and publish-subscribe interactions	Processes are formed dynamically therefore never completely sure what services will participate.

By creating a combination of services at run time based on consumer requests, the system can form topologies not specifically planned at design time.

Impact of Conjunctive Composition

The enabling of features necessary for Conjunctive Composition challenges the nature of a requirements base for SOA delivery – which requirements, the ones at time of initial SOA delivery, or those used at the time of conjunctive composition, should be used as the basis for measuring assurance and trustworthiness? Each feature results in a challenge to providing a secure and information assured environment.

TABLE C-2. Conjunctive Composition Security and IA Challenges

Feature	***Security and IA Challenges***
Service contracts	• Designing and defining voluntary agreements that mutually bind the participants to authorizations, obligations, and modes of interaction.
Dynamic Process Formation	• Determining the value proposition for the consumer and then identifying and assembling services that will form a virtual service for the end consumer.
Composite services	• Centralized reuse repository and effective reuse management
Application-neutral design	• Proper domain analysis, and good ontology specification and management
Consumer and Provider Metadata	• Proper domain analysis, and good ontology specification and management • How will the metadata attributes attached to a service be inseparable from the service?
Concurrently developed	• Application context in the metadata, not the software component – how do you authenticate and authorize to Metadata?
Interoperable across a heterogeneous environment	• What represents the correct information context? • How do you assure a consistent specification of information assurance across different environments and different stakeholders? • What does a cross-domain service require of another service to accomplish information sharing across domains? • How will the transformation be handled? Specifically, how to manage and secure a GIG which is evolving and only partially SOA-compliant?
Reusable artifacts	• Centralized reuse repository and effective reuse management
Network addressability	• If services can be pulled from 'anywhere', how do we restrict service requests to only pull 'approved' service for the data sensitivity level?

Additional SOA Impacts

SOA impacts different areas with regard to security. The following highlights relevant questions for each area:

Impact of SOA on Roles and Responsibilities

- What will the service provider, the infrastructure, and the client be responsible for providing to secure the SOA?
- How roles of service consumers with services are adjudicate?
- How are permissions assigned at run time?

Impact of SOA on Threats and Threat Specifications

Table C-3 illustrates how the threat model is extended to accommodate SOA IA threats. Here are some questions:

- What specific threats are introduced by the addition of a SOA architecture and what are the specific services and protection points to address those threats?
- How should the system respond with denial of rejected requests?

TABLE C-3. Extending the Threat Model to accommodate SOA IA Threats

	Threats	***Services***
1	User Impersonation Service Impersonation User exceeding assigned authorization	• User Identification • User Authentication • User Authorization
2	Undesired Use of an Object Implementation Request/Response Repudiation Disclosure of "Eyes-only" data	• Application-Layer Access Control • Non-Repudiation • Security Audit Logging • Data Protection
3	Unprotected Security-Unaware Applications Unwanted Revelation of Client Machine Existence	• Client-side Object Invocation Access Control • Data Protection
4	Object Masquerade Client Masquerade Object Mis-use of User Authorizations IOR tampering Disclosure of Request Contents Modification/Destruction of Request Contents	• Authentication Between Client and Object • Encryption Between Client and Object • Delegation Controls • Security Audit Logging
5	Network Eavesdropping Message Tampering Inability to cross network boundaries (e.g., firewalls)	• Transport Encryption • IIOP Traversal of Firewalls
6	Unprotected Security-Unaware Applications Too many object interfaces and implementations to manage individually	• Server-side Object Invocation Access Control • Security Policy Domains
7	Unauthorized Disclosure of Specific Information to Client Request/Response Repudiation Protection of "Eyes-only" data	• Application-Layer Access Control • Non-Repudiation • Security Audit Logging • Data Protection
8	Message Content validation Message identity guarantee Is the message from a 'live' and valid service?	

Impact of SOA on Protection Points

- How will authorizations be applied when authorization repositories must be accessed across domains?
- How will the service repository be "self protecting"? What are the requirements for the repository? (Is it higher than the resulting network?)
- Service requestors (consumers of a service) can be end users or other Components or services. How do you assure the identity of the service requestor?
- Is the Service Registry the intermediary that must (a) determine whether a request can be satisfied for a service based upon the security profile of the service requestor and the security level(s) of the possible services, and (b) are multiple security levels of a requested service available to satisfy the requests of multiple levels of security for service requestors?

Impact of SOA on Physical Environment

- What has to be added to the physical environment to ensure SOA security, given the above?
- What can be reused, maybe with additions or adjustments, in the physical environment to ensure SOA security, given the above?
- How will SOAs perform in areas of limited bandwidth? (front line, wireless, dial up, etc)

Impact of SOA on Situational Awareness

Situational awareness, in a systems context, is the capability to determine at run time if the combination of requests and peer services is a permissible combination in view of security restrictions. This assumes that the system will apply a set of rules at service request time that derive from the nature of the requests. The system may respond by providing more limited access to specific services than requested. There are issues are how to notify the requestor and how to avoid breakage.

- How will we maintain Situational Awareness of SOA functionality and performance?
- What contributes at any one point in time to the services that represent a particular situational context?

Extending the SOA UML Profile for Security

In Appendix B a UML profile for effective and consistent description of a SOA was overviewed. As currently specified, this profile will accurately describe in a measurable way the dynamic and conjunctive characteristics of a SOA delivery, but not the specifications necessary to describe a secure and assured solution.

However, a companion effort is underway that will *extend* – that is, add new components to the SOA UML Profile – and *specialize* – that is, add attributes and methods to existing profile classes – so that security and assurance characteristics are accommodated. Figure C-1, below, contains a snippet of part of the SOA UML Profile describing the relationship of a service – a well scoped and specifically defined outcome – to the mechanism that enables the service: an interface. It also describes a relationship to the Community of Interest that would subscribe to the service and use the Interface. A *specialization* of this part of the SOA UML Profile would be to add necessary attributes to SERVICE and INTERFACE to accommodate security contexts. An *extension* to this part of the SOA UML Profile would add a parametric model on the association between SERVICE and INTERFACE and SERVICE and COMMUNITY of INTEREST to describe how these relationships are situationally authorized.

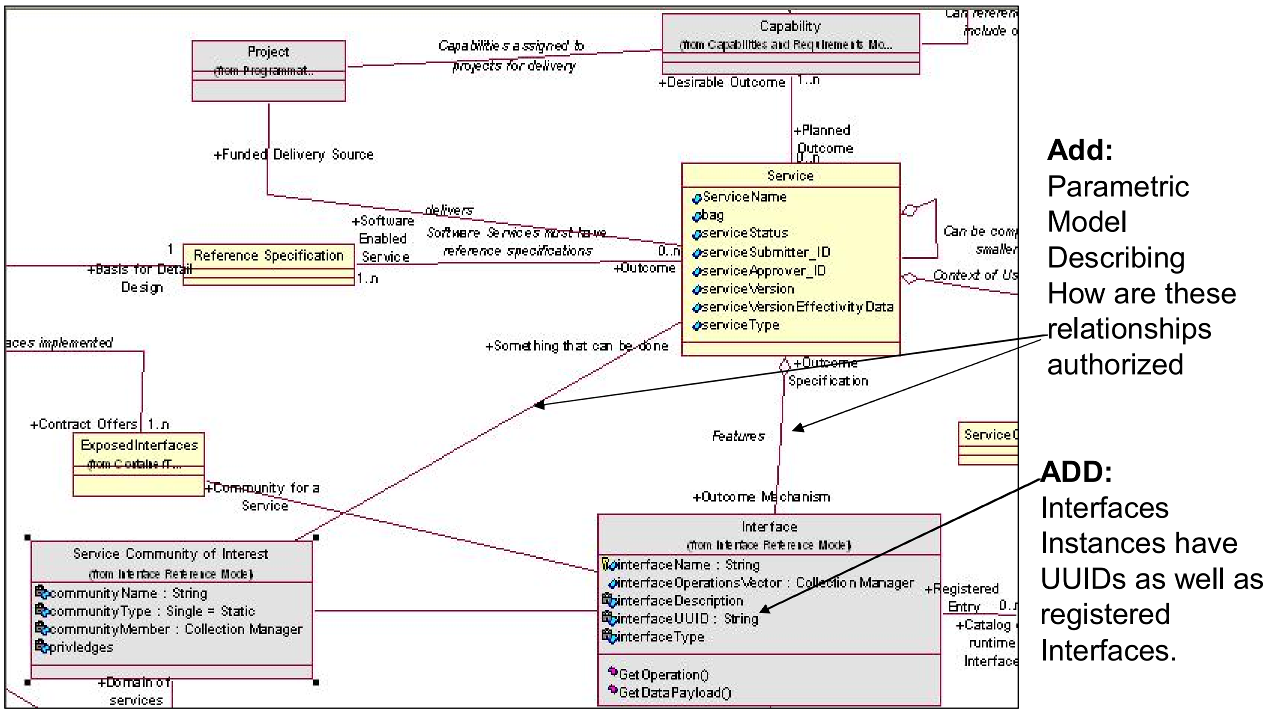


FIGURE C-1
Extending the SOA UML Profile to Address Security and Assurance

The implication of this is that all of the impacts on security and assurance resulting from a SOA design to delivery implementation overviewed in the previous section would be resolved into to the SOA UML Profile.

SOA IA Methodology Impacts

The SOA Software Engineering Process will naturally accommodate security and assurance conceptual integrity along with all the other classes and associations used to accurately describe a SOA by:

1. Adding security and assurance *extensions* and *specializations* to a SOA UML Profile, and
2. Incorporating the SOA UML Profile into the Methodology used for SOA solution delivery

In other words, security and assurance are just another set of characteristics that must be accommodated and the long desired for ‘security must be designed-in and not pasted-on’ is now realized.

Conclusion

- SOA adds a number of new features change the nature of systems delivery.
- These additional characteristics results in a significant impact on the foundations of software systems engineering.

- Using a method and process that is Service aware, such as Feature Driven Development coupled with SCRUM-based processes will accommodate these new features.
- Further, these features can be well described and used as a basis for deriving a UML Profile that accurately and consistently describes SOA in a measurable way. This will improve the accuracy, consistency, and interoperability of Service-based solutions.
- But unfortunately, the features of SOA results in the introduction of a number of new threats and systemic characteristics that challenges the traditional approach to security and assurance.
- However, by effective description of these threats and characteristics, they can be added to a SOA UML Profile so that—
    - Tool vendors can build tools to accommodate not just the specification and delivery of SOA, but the security and assurance characteristics as well.
    - Service providers and provide solutions that are consistent with the integrate specification thereby improving interoperability and conjunctive composition. And reuse is improved as well.
    - Methodologies that assume a SOA UML Profile and are derived from the features noted above will naturally accommodate security and assurance issues.